CS 241 – Week 2 Tutorial

Binary Numbers and Assembly Language Basics

Spring 2017

1 Binary

1.1 Converting from Binary to Decimal

The computer is a very powerful tool that is constantly evolving. In this course, we will introduce the idea of how a program runs on a computer. To begin, we will review how data is represented, i.e. binary representation.

- Recall that binary data has no meaning on its own: we need to be told (or come up with) an interpretation for it. Give five different possible ways we could interpret 0101.
- Convert the 8-bit binary number 01101001 into decimal.

We can convert positive numbers into binary by repeated division by 2. For example, to convert 23 into binary:

- $23/2 = 11$ remainder 1
- $11/2 = 5$ remainder 1
- $5/2 = 2$ remainder 1
- $2/2 = 1$ remainder 0
- $1/2 = 0$ remainder 1

Then reading digits from bottom to top, we get $23_{10} = 10111_2$. To verify this, we can work backwards: $10111_2 = 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = 23$. Leading digits are just 0, so in 8-bit binary we would write 00010111.

- Convert the following numbers into unsigned 8-bit binary:
  1. 35
  2. 216

1.2 Two's Complement

If we also want to be able to represent negative numbers as well as positive ones, we use an encoding called "two's complement". The two's complement encoding can feel somewhat arbitrary, but is based on the important property that $x + (-x) \equiv 0 \mod 2^b$. You can find the two's complement encoding of a number in b-bit binary in a number of ways. Two simple ones are:

1. Encode the number as if it were unsigned (so if you want to encode -123, encode 123, then apply the two's complement algorithm presented last week: flip all bits and add 1.
   - Why does this work? Call x^* the result of flipping all the bits of x (so for example, if $x = 1011$ then $x^* = 0100$). Then $x + x^* = 1111 = 2^b - 1$. So since we picked $-x = x^* + 1$, we get $x + (-x) = x + x^* + 1 = 2^b - 1 + 1 = 2^b \equiv 0 \mod 2^b$.
2. Instead of encoding $-x$, encode $2^b - x$.
   - Why does this work? $x + (-x) = x + 2^b - x = 2^b \equiv 0 \mod 2^b$.

Try the following in 8-bit binary:

1. Encode -12 using the first technique.
2. Encode -123 using the second technique.
3. Convince yourself that, for example, $--15 = 15$ by either two's complement technique listed above. What is $--128$?

Side Question: Why do we use two's complement instead of sign magnitude?

2 Assembly Language

2.1 Assembly Language Instructions

Assembly languages are simple programming languages which let us transfer values between memory and registers, and perform simple arithmetic and logic operations. We will use MIPS assembly language in this course. For example, `add $3, $1, $2` means "add together the values in registers 1 and 2 and place the result in register 3." Note that the destination comes first, just like in C++, e.g. `r3 = r1 + r2`. Note that most instructions cannot take constant values as arguments!

2.2 Registers

- Registers in MIPS each hold 32 bits of information, and can be thought of as the variables that we are given to work with in our program.
- Some registers are special:
  - $0 always contains the value 0 and the contents of $0 cannot be modified in any way.
  - $31 is reserved for the return address.
  - We will make $3, $29, and $30 special by convention in this course.

  Try get in the habit of using them only for their intended purpose, which will explained later on in the course.

2.3 Constant values

Most MIPS instructions in this course deal with registers instead of literal values.

We solve this using the load immediate and skip (lis) instruction with the .word directive to load a value into a register. For example,

```
lis $5
.word 7
```

This pair of instructions stores the value 7 into $5.

2.4 Assembling assembly language

The process of converting assembly language into code the computer can process (i.e. machine code) is called assembling. The process of turning assembly language into code the computer understands (i.e. binary) is called assembling. Soon were going to automate this process, but for now well do it by hand. The MIPS assembly language reference sheet (https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsref.pdf) gives a template for how to assemble each 32-bit instruction.

Assemble the following program by first writing out its binary representation, then converting it to a hexadecimal representation, and finally using `cs241.wordasm` to make it executable. What does it do? Run it with `mips.twoints` to verify that it behaves as you'd expect.

```
lis $5
.word 7
slt $6, $1, $5
beq $6, $0, 1
add $1, $1, $5
jr $31
```